Zum Auftakt booten (oder reaktivieren) Sie die Ulix-Devel-VM und führen in der Shell den Befehl `update-ulix.sh` aus. Damit laden Sie die Dateien herunter, die Sie für das Bearbeiten der Projektaufgaben benötigen.

Im Ordner `~/ulix/` (`/home/ulix/ulix/`) finden Sie die Ulix-Version 0.10 – das ist nicht der bisher aus dem Übungsbetrieb bekannte, reduzierte Ulix-Code, sondern der „echte“ Ulix-Code, der deutlich mehr Features als die Versionen aus den Übungen bietet. Das zugehörige Literate Program `ulix-book.nw` enthält am Ende ein Kapitel (*chapter*) *Praktikum*: Hier ergänzen Sie für jede bearbeitete Aufgabe eine *section* mit der Aufgabennummer als Überschrift (`\section{Aufgabe X}`) .

Die Datei ist eine ältere Version des Lehrbuchs *The Design and Implementation of the ULIX Operating System*, das Sie zu Beginn der Veranstaltung als PDF-Datei erhalten haben und das Ulix 0.13 beschreibt.

Bei vielen Aufgaben wird es nicht zu vermeiden sein, auch den bereits vorhandenen Code anzupassen, etwa um in der Kernel-Initialisierung den „Aufruf“ eines zusätzlichen Code-Chunks einzubauen. Wenn Sie das tun, weisen Sie im Praktikumskapitel an geeigneter Stelle auf die Ergänzung hin.

Für alle Aufgaben zu System Calls können Sie in den alten Übungsaufgaben nachsehen, wie die Übergabe von Parametern und die Rückgabe des Rückgabewerts funktionieren – das Prinzip ist immer dasselbe (und es funktioniert im „großen“ Ulix genau wie in den bisher benutzten kleinen Versionen).

Wenn Ihnen die Lösung einer Aufgabe nicht oder nicht vollständig gelingt, können Sie diese trotzdem abgeben – nehmen Sie dann Hinweise auf das Fehlverhalten und evtl. Vermutungen zu Fehlerursachen in die Literate-Programming-Dokumentation auf.

## Wichtig:

1. Sie können beim Lösen der Aufgaben *mit Einschränkungen* **zusammenarbeiten**. Wenn Sie eine Zweiergruppe bilden möchten, sollten Sie bei der gemeinsamen Arbeit an zwei separaten Rechner entwickeln und sich dann nur zu Detailfragen besprechen („Wie hast Du XYZ gelöst?“). Sie können mich außerdem während der Präsenztermine oder nach den Terminen per E-Mail um Hilfe bitten, wenn Sie an einer Stelle nicht weiter kommen.
2. **Das Literate Program** (das Code und Dokumentation enthält) **muss jeder Teilnehmer individuell anfertigen.** Identische bzw. strukturgleiche Dokumentation in den Abgaben von zwei oder mehr Teilnehmern ist ein nicht widerlegbares Indiz für unerlaubte Zusammenarbeit und führt zum Nichtbestehen.
3. Bitte berücksichtigen Sie beim Programmieren, dass **50 % der erreichbaren Punkte für die Qualität der Dokumentation im Literate-Programming-Stil** vergeben werden. Erstellen Sie also nicht zunächst das Programm und ergänzen dann Dokumentation – dieser Ansatz führt meistens zu schlechtem LP-Stil und fällt auf.

   Ein sauber im LP-Stil geschriebenes Programm mit kleineren Fehlern im Code wird typischerweise eine bessere Bewertung erhalten als ein Programm mit perfektem Code und nachträglich ergänzter Dokumentation.

Es folgen Hinweise zu den verschiedenen Aufgabentypen.

## Pflicht- und Wahlaufgaben

Es gibt zwei Arten von Aufgaben:

- Die ersten Aufgaben **P1**–**P3** sind Pflichtaufgaben, sie dienen im Wesentlichen dazu, einen Einstieg in die Arbeit zu finden. Für die **P**-Aufgaben gibt es insgesamt **20 Punkte**.
- Danach folgen Wahlaufgaben **W1**–**W6**. Hier können Sie nach Belieben mehrere Aufgaben auswählen, so dass Sie im W-Bereich auf **60 bearbeitete Punkte** kommen. Bei jeder Aufgabe steht hinter dem Titel eine Punktzahl. Aufgaben, die aus mehreren Teilaufgaben (a, b, c, …) bestehen, können Sie immer nur vollständig auswählen.

  Beispiel: Sie bearbeiten W1, W2 (je 10 Punkte) und W5 (40 Punkte), Summe 60. Oder: Sie bearbeiten W5 und W7 (40 + 20 = 60 Punkte).

  Es ist *nicht* möglich, mehr Wahlaufgaben zu bearbeiten und alle nur teilweise zu lösen – Sie müssen eine Auswahl treffen.

  Die mit zwei Sternchen gekennzeichneten Aufgaben W5 bis W7 sind etwas anspruchsvoller als die übrigen Aufgaben.

Insgesamt sind also 20 + 60 = 80 Punkte erreichbar. (Wenn Sie bei der Auswahl einen Fehler machen und zu viele Aufgaben bearbeiten, frage ich per Mail nach, welche Sie in der Wertung haben wollen.)

## Abgabe, Deadline

Schicken Sie ein `Zip`- oder `tar.gz`-Archiv mit den Dateien `ulix-book.nw`, `ulix-book.pdf` und allen selbst erstellten Programmdateien aus `lib-build/tools/` per E-Mail an mich. Bei der Wertung des Literate Programming berücksichtige ich nur Textteile, die im hintersten Kapitel (*Anhang C: Praktikum*, ab Seite 431) stehen. Deadline für die Abgabe ist am Sonntag, **07.02.2015** (23:59 Uhr).

## *Pflichtaufgaben*

# P 1. Überblick über Quellcode und Build-Prozess (0 Punkte)

Zum Auftakt machen Sie sich mit der Verzeichnisstruktur des Ulix-Quellcodes und mit dem Build-Prozess vertraut. Für diese Aufgabe gibt es keine Punkte, sie dient nur der Orientierung.

Direkt in `~/ulix/` (`/home/ulix/ulix/`) liegt die wichtigste Datei, das Literate Program `ulix-book.nw`. Wenn Sie in diesen Ordner wechseln, können Sie die automatische Übersetzung mit `make` und die Ausführung von Ulix mit `make run` anstoßen (wie in den Übungen). Über `make pdf` erzeugen Sie die formatierte Dokumentation (`ulix-book.pdf`). Mit ca. 450 Seiten hat sie etwa den zehnfachen Umfang der bisher in den Übungen gesehenen Dokumentation. Es ist nicht nötig, all diese Seiten zu lesen, aber Sie sollten einen Überblick über den Aufbau gewinnen, damit Sie relevante Programmstellen schnell finden können – in der Quelldatei oder in der PDF-Datei.

Am schnellsten erhalten Sie einen Überblick über den Code, indem Sie die PDF-Datei ansehen – darin können Sie auf Code Chunks klicken, um zu den jeweiligen Code-Stellen zu kommen. Die Definition der Quellcode-Datei `bin-build/ulix.c` beginnt auf Seite 25 im Code-Chunk *<ulix.c>*, die Initialisierung des Kernels endet mit dem Wechsel in den User Mode und dem Start der Shell (in *<start shell>*). Assembler-Code müssen Sie im Rahmen des Projekts nicht erstellen; wenn Sie einen Blick hinein werfen möchten, finden Sie ihn in der Datei `bin-build/start.asm`.

Alle Ulix-Anwendungen nutzen eine User-Mode-Bibliothek, die in Kapitel 21 (ab Seite 405) implementiert ist. Sie ist nur sehr unvollständig dokumentiert und bietet noch nicht allzu viele Features; einige werden Sie im Rahmen der Projektaufgaben ergänzen. Es gibt hier nur zwei Code Chunks *<ulixlib.c>* und *<ulixlib.h>*, welche beim `make`-Aufruf in die Dateien `lib-build/ulixlib.c` und `lib-build/ulixlib.h` extrahiert werden.

Ulix-Programme finden Sie im Unterordner `lib-build/tools`. Wenn Sie ein neues User-Mode-Programm erzeugen möchten, erzeugen Sie in diesem Ordner eine neue C-Datei – beim `make`-Aufruf werden automatisch alle C-Programme in diesem Ordner übersetzt, und die entstehenden Programmdateien landen im Disketten-Image, das Ulix als Root-Dateisystem verwendet. Wenn Sie im Rahmen einer Projektaufgabe ein solches Programm erstellen, müssen Sie den Code von Hand aus dem C-Programm nach `ulix-book.nw` kopieren. Das erwähnte Disketten-Image ist `bin-build/minixdata.img`. Um neue Ordner oder Dateien auf das Image zu kopieren, können Sie diese im Ordner `lib-build/diskfiles` ablegen – beim nächsten Lauf von `make` landen die Dateien dann im Image. Wollen Sie Dateien vom Image löschen, können Sie das unter Ulix mit dem `rm`-Befehl erledigen.

Werfen Sie im laufenden Ulix-System einen Blick in den Ordner `bin/` und probieren Sie einige der dort liegenden Programme aus. `cat`, `clear`, `cp`, `diff`, `hexdump`, `ps`, `rm`, `touch` und `vi` funktionieren ähnlich wie die entsprechenden Linux-Anwendungen, die übrigen Tools haben keine sinnvolle Aufgabe und sind für Tests von Ulix-Features gedacht.

# P 2. Hello World (12 Punkte)

In dieser Aufgabe wiederholen Sie im Wesentlichen Arbeitsschritte, die Sie bereits aus den Übungsaufgaben kennen; es geht hier darum, sich an die neue und komplexere Code-Umgebung zu gewöhnen. Ziel ist, einen neuen System Call ins System einzubauen und aus dem Hauptprogramm heraus aufzurufen.

Anders als bei den Ulix-Versionen im Übungsbetrieb hat das richtige Ulix ein Dateisystem, das im Ordner `bin/` ausführbare Ulix-Programme enthält. Der Prozess der Übersetzung und Aufnahme ins Dateisystem-Image ist durch das Makefile automatisiert. (Sie müssen also nicht mehr, wie in Übung

12–14, manuell Listen mit Hexzahlen in den Ulix-Quellcode einbauen.)

**a)** Schreiben Sie eine Kernel-Funktion

```
void u_hello (int n)
```

die `"Hello World - %d\n"` und über den Format-String das Argument `n` ausgibt. Den Prototyp nehmen Sie in den Code Chunk <*function prototypes*> auf, die Implementierung in <*function implementations*>.

**b)** Schreiben Sie einen Syscall-Handler

```
void syscall_hello (context_t *r)
```

welcher aus `r` ein Integer-Argument extrahiert und damit die Funktion `u_hello ()` aufruft. Auch der Syscall-Handler benötigt wieder Prototyp und Implementierung in den passenden Code Chunks.

**c)** Definieren Sie für den Syscall eine Syscallnummer, z. B. mit

```
#define __NR_hello 777
```

und tragen Sie mit `install_syscall_handler ()` den neuen Syscall in die Syscall-Tabelle ein. Sie können den Aufruf in den Code Chunk <*initialize syscalls*> aufnehmen.

**d)** Schreiben Sie eine User-Mode-Bibliotheksfunktion `void hello (int n)`, welche via `syscall2 ()` den Syscall 777 ausführt. Auch Bibliotheksfunktionen benötigen Prototyp und Implementierung, dafür sind aber andere Code Chunks zuständig.
Der Protoyp gehört nach <*ulixlib.h*>, die Implementierung nach <*ulixlib.c*>.

**e)** Übersetzen Sie mit `make` den geänderten Code, um sicherzustellen, dass er korrekt kompiliert.

**f)** Wechseln Sie in den Unterordner `lib-build/tools/` und ändern Sie die dort bereits vorhandene Programmdatei `hello.c` ab: In der `main()`-Funktion testen Sie `hello ()`, z. B. Mit `hello (5)`; die übrigen Befehle in `main()` entfernen Sie. Wechseln Sie dann wieder in den Hauptordner zurück (nach `/home/ulix/ulix`) – wenn Sie jetzt `make` eingeben, wird das neue Programm übersetzt. Starten Sie mit `make run` das System und testen Sie den Befehl `hello`. Das Programm `hello` liegt (unter Ulix) im Ordner `bin/`.

(Wichtig ist in allen Ulix-Programmen, dass Sie die Header-Datei `../ulixlib.h` mit

```
#include "ulixlib.h"
```

einbinden und – falls Sie neben `main()` noch andere Funktionen im Programm implementieren, immer mit dem Code von `main()` starten. Dann müssen Sie die übrigen Funktionen durch Angabe des Prototyps vorausdeklarieren. Ein Beispiel dafür finden Sie in `diff.c`: Dort wird `memcmp()` vorausdeklariert und erst nach `main()` implementiert. Außerdem muss jedes Ulix-Programm explizit mit `exit(0)` beendet werden – ein Verlassen von `main()` mit `return` verursacht Fehlermeldungen in Ulix.)

# P 3. Status mit Strg-Esc (8 Punkte)

Wenn Sie im laufenden Ulix [Umschalt-Esc] drücken, landen Sie in der Kernel-Shell (und alle Prozesse werden angehalten). Geben Sie dort `exit` ein, gelangen Sie zurück in den User Mode, und die Prozesse laufen wieder weiter.

Der Wechsel in die Kernel-Shell passiert im Tastatur-Interrupt-Handler `keyboard_handler()` (ab Seite 183). Schauen Sie sich an, auf welche Weise der Handler auf [Umschalt-Esc] reagiert.

Fügen Sie nun an geeigneter Stelle im Code einen Aufruf des Code Chunk <*project keyboard handler*> ein. Diesen Code Chunk können Sie im Projektkapitel implementieren. Er soll analog zum

Aufruf der Kernel-Shell eine Funktion `void status_break()` aufrufen, wenn Sie [Strg-Esc] drücken.

Diese gilt es nun zu implementieren. Ulix zeigt ohnehin ständig am unteren Rand eine Statuszeile an. Die von Ihnen zu schreibende Funktion `void status_break ()` soll die folgenden Aufgaben ausführen:

- Interrupts deaktivieren mit <*disable interrupts*>
- einen maximal 80 Zeichen langen Informations-String zusammen bauen, der die aktuelle Thread-ID, die aktuelle Address-Space-ID, die freien Frames, die Anzahl aller Threads, die Anzahl aller Threads in der Ready Queue, die Ulix-Versionsnummer enthält.
- diesen String mit `set_statusline()` (siehe S. 381) anzeigen
- ca. eine Sekunde warten (passende Zählschleife basteln)
- Statuszeile wieder auf Standardinhalt (`UNAME`) setzen
- Interrupt wieder aktivieren mit <*enable interrupts*>

## *Wahlaufgaben*

*Wählen Sie hier mehrere Aufgaben aus, die zusammen **60 Punkte** wert sind.*

*Die Aufaben W 5 ** bis W 7 ** sind etwas schwieriger als die übrigen und deshalb mit ** gekennzeichnet.*

## W 1. Mehr Shells (10 Punkte)

Ulix verwendet zehn virtuelle Konsolen, die Sie mit [Alt-1] bis [Alt-9] und [Alt-0] erreichen können – auf den ersten fünf Konsolen laufen Shells. Passen Sie den Ulix-Code so an, dass acht Shells (auf den ersten acht Konsolen) verfügbar sind. Finden Sie dazu zunächst heraus, wo (im Kernel, in der Bibliothek?) die Konsolen aktiviert werden.

Zum Dokumentieren Ihrer Änderungen fügen Sie im *Projekt*-Kapitel neue Code Chunks (mit von Ihnen vergebenen Chunknamen) ein, welche den geänderten enthalten. An den Originalstellen im Code löschen Sie den ursprünglichen Code und verweisen auf den neuen Code Chunk.

## W 2. Syslog im Dateisystem (10 Punkte)

Diese Aufgabe kommt *ohne* System Calls aus. Schreiben Sie zwei neue Kernel-Funktionen

```
void syslog_open (char *filename);
void syslog_write (char *msg);
```

mit denen Sie eine Syslog-Datei öffnen und Strings hinein schreiben können. Dabei soll `syslog_write` dem übergebenen String immer den aktuellen Wert von `system_ticks` und ein Leerzeichen voranstellen sowie ein Zeilenumbruchzeichen (`\n`) am Ende anhängen. Zum Formatieren eines Strings können Sie die Kernel-Funktion `sprintf ()` verwenden, die wie die gleichnamige Linux-Funktion arbeitet.

Suchen Sie sich dann eine Kernel-Funktion, die häufig aufgerufen wird, und ergänzen Sie dort einen Aufruf der Form

```
syslog_write ("funktionsname: entering");
```

am Anfang der Funktion.

An geeigneter Stelle (während der Initialisierung des Kernels) müssen Sie dann `syslog_open()` aufrufen. Falls es zu Aufrufen von `syslog_write()` kommt, bevor die Logdatei geöffnet wurde, soll die Funktion einfach mit `return` zurückkehren.

Testen Sie im neu kompilierten System, ob Sie mit mehrfacher Eingabe von `cat logdateiname` sehen können, wie neue Einträge hinzu kommen.

## W 3. Boost: Nur diesen Prozess ausführen (20 Punkte)

Der Scheduler, der von `timer_handler()` alle zwei Ticks aufgerufen wird, schaltet jeweils auf den nächsten Prozess in der Ready Queue um. In dieser Aufgabe geben Sie einem Prozess die Möglichkeit, dieses Umschalten zu verhindern: Über die Funktion `void boost (int n)` kann er eine globale Variable (global im Ulix-Kernel) setzen, und im `timer_handler` (in einer der *<timer tasks>*) soll dann geprüft werden, ob diese einen Wert != 0 hat. Wenn ja, wird der Wert um 1 erniedrigt, und es gibt keinen Context Switch.

**a)** Deklarieren Sie an geeigneter Stelle eine globale Variable `int boost_count`, die Sie auf 0 initialisieren.

**b)** Verschieben Sie den Code Chunk *<timer tasks>*, der `scheduler ()` aufruft, in den Praktikum-Anhang, damit Sie die dort nötigen Änderungen dokumentieren können; es geht um diesen Chunk:

*<timer tasks>*=
```
// Every 5 clocks call the scheduler
if (system_ticks % 5 == 0) {
 [...]
   scheduler (r, SCHED_SRC_TIMER);  // defined in the process chapter
 [...]
```

**c)** Passen Sie die Fallunterscheidung an; es muss zusätzlich die Bedingung `boost_count == 0` erfüllt sein. Außerdem brauchen Sie einen `else`-Fall, in dem Sie `boost_count` dekrementieren.

**d)** Schreiben Sie einen Syscall-Handler

```
void syscall_boost (context_t *r);
```

der aus dem richtigen Register den übergebenen Wert liest und ihn nach `boost_count` schreibt (falls er positiv oder 0 ist). Definieren Sie eine Sycall-Nummer-Konstante `__NR_boost` (wobei Sie eine noch nicht vergebene Nummer verwenden) und tragen Sie in den richtigen Code Chunk einen Aufruf von `install_syscall_handler ()` ein.

**e)** Schreiben Sie eine User-Mode-Bibliotheksfunktion

```
void boost (int n);
```

welche mit Hilfe von `syscall2 ()` den Syscall auslöst. Sie können sich an der Implementierung von `open ()` orientieren.

**f)** Schreiben Sie ein kleines User-Mode-Programm, mit dem Sie den Effekt von `boost ()` überprüfen können.

(In dieser Aufgabe weichen wir vom Standard-Prozedere ab: Normal gäbe es hier noch eine Kernel-Funktion `u_boost`, die vom Syscall-Handler aufgerufen würde – da diese aber einfach

```
void boost (int n) { boost_count = n; }
```

wäre, ist es unsinnig, daraus eine Funktion zu machen.

## W 4. Unix-Name: uname() (20 Punkte)

Alle Unix-Systeme bringen ein Kommandozeilenprogramm `uname` mit, das Auskunft über das laufende Betriebssystem gibt. Das Programm nutzt dazu einen System Call mit demselben Namen.

**a)** Lesen Sie die Manpage zur Linux-Bibliotheksfunktion `uname` durch (`man 2 uname`; ohne die `2` erhalten Sie die Manpage des Shell-Kommandos `uname`), beachten Sie dabei insbesondere die Datenstruktur `struct utsname`.

**b)** Definieren Sie in Ulix einen neuen Typ `struct utsname` analog zur Definition aus der Manpage, lassen Sie aber den Eintrag `char domainname[]` weg.

**c)** Implementieren Sie eine neue Funktion `u_uname ()`:

```
int u_uname (struct utsname *buf);
```

welche den übergebenen Buffer mit geeigneten Werten (`sysname`: Ulix, `nodename`: ulixmachine, `release`: 0.10 bzw. letzte vier Zeichen in `UNAME-#define`-Konstante, `version`: `BUILDDATE-#define`-Konstante, `machine`: i386) füllt.

**d)** Schreiben Sie einen Syscall-Handler `syscall_uname ()` mit folgender Signatur:

```
void syscall_uname (context_t *r);
```

Er soll die Funktion `u_uname ()` aufrufen.

Denken Sie auch daran, den neuen Syscall-Handler zu registrieren. Die Syscall-Nummer ist in `ulix-book.nw` bereits in der Zeile

```
#define __NR_uname              122
```

deklariert.

**e)** Ergänzen Sie die User-Mode-Library um die schon im Kernel verwendete Typdeklaration von `struct utsname` sowie eine Funktion

```
int uname (struct utsname *buf);
```

welche mit Hilfe von `syscall2 ()` den Syscall auslöst. Sie können sich an der Implementierung von `open ()` orientieren.

**f)** Entwickeln Sie ein neues User-Mode-Programm `uname` (im Ordner `lib-build/tools/`), das die `uname`-Funktion verwendet und die Ergebnisse in ähnlicher Form wie das `uname`-Programm unter Linux (beim Aufruf mit Option `-a`) ausgibt. Testen Sie, dass sich das Programm unter Ulix starten lässt und dass es die richtigen Informationen ausgibt.

## W 5 **. Ridiculously Simple Filesystem (40 Punkte)

Das Ridiculously Simple Filesystem (RSF) arbeitet mit einer Standardsektorgröße von 512 Byte, und ein Image hat folgenden Aufbau:

- Sektor 0 enthält die FAT (File Allocation Table), die bis zu 32 FAT-Einträge der Größe 16 Byte enthält (16 x 32 = 512).
- Jeder FAT-Eintrag enthält die folgenden Daten:
  - Bytes 0–11: Dateiname. Wenn der Dateiname weniger als 12 Zeichen lang ist, enthalten die restlichen Bytes `\0`. (Achtung: Bei Dateinamen der Länge 12 gibt es hier keine `\0`-Terminierung des Namens.)
  - Bytes 12–13: Dateigröße in Byte, die maximale Dateigröße auf einem RSF-Medium ist also $2^{16}$ Byte = 64 KByte.

    - Bytes 14–15: Sektornummer des ersten Datenblocks der Datei
- Alle Dateien müssen zusammenhängend im Image gespeichert werden, weil in der FAT nur der Startsektor vermerkt wird. Die erste Datei beginnt in Sektor 1, gleich hinter der FAT. Unbenutzte FAT-Einträge erkennen Sie daran, dass das erste Byte des Eintrags `\0` ist. Hinter einem unbenutzten Eintrag dürfen keine benutzten mehr folgen.

**a)** Entwickeln Sie ein kleines (Linux-) Kommandozeilentool `mkfs.rfs`, das Sie in der Form `./mkfs.rfs disk.img file1 file2 file3 …` aufrufen, um die Dateien `file1`, `file2`, `file3`, … ins Image `disk.img` zu kopieren. Alle Quelldateien müssen im aktuellen Verzeichnis liegen (das Tool kommt nicht mit Pfadangaben zurecht); die Zieldatei (im Beispiel `disk.img`) darf aber auch in einem anderen Ordner liegen.

**b)** Sie werden nun Ulix um einen RSF-Treiber erweitern. Entwickeln Sie dazu die folgenden Funktionen, die alle davon ausgehen, dass die zweite virtuelle Platte (`hdb.img`) ein mit `mkfs.rsf` erzeugtes RSF-Image enthält:

- `void rsf_ls()`: Gibt eine Liste aller Dateien wie folgt (Größenangabe in Byte) aus

```
filename       size  sectors
test.txt         30  0001-0001
test2.txt     32768  0002-0065
Makefile         12  0066-0066
```

- `int rsf_open(char *filename)`: eine vereinfachte Version des Datei-Öffnens. Es kann immer maximal eine Datei geöffnet sein. Je nach Situation reagiert `rsf_open` wie folgt:
    - Es ist bereits eine Datei geöffnet → Abbruch, Rückgabewert -1.
    - Es ist noch keine Datei geöffnet, aber `filename` existiert nicht auf dem Datenträger → Abbruch, Rückgabewert -1.
    - Es ist noch keine Datei geöffnet, und `filename` existiert → Erfolg. Die Funktion merkt sich in einer globalen Variable den Namen der Datei. Rückgabewert 0.
- `int rsf_readsector(int fd, int secno, char *buf)`: liest einen Sektor aus der geöffneten Datei (falls `fd == 0`). Wenn keine Datei geöffnet ist oder `fd != 0` gilt, Abbruch mit Rückgabewert -1. Ansonsten: Rückgabewert = Anzahl der gelesenen Bytes. Vorsicht: Beim letzten Sektor einer Datei ist das oft ein Wert < 512.
- `void rsf_close(int fd)`: schließt die Datei (falls eine offen war); `fd` ist hier beliebig.

Es gibt keine Funktion zum Schreiben, weil RSF ein Read-only-Dateisystem ist, vergleichbar mit dem ISO-Dateisystem auf CDs und DVDs.

*Hinweis*: Damit Ulix auf ein zweites Platten-Image zugreifen kann, müssen Sie in `bin-build/Makefile` im Target `run` (in der letzten Zeile) den Aufruf von `qemu` anpassen, Sie brauchen die Zusatzoption `-hdb hdb.img` (wenn das Image `hdb.img` heißt und auch in `bin-build/` liegt).

**c)** Ergänzen Sie System Calls und User-Mode-Bibliotheksfunktionen, über die Sie `rsf_open`, `rsf_readsector` und `rsf_close` in User-Mode-Programmen verwenden können, und schreiben Sie ein Testprogramm.

**d)** Aufbauend auf `rsf_readsector()` schreiben Sie nun Funktionen `rsf_lseek()` und `rsf_read()`, die wie `lseek` und `read` arbeiten (siehe manpages); hierbei ist das Lesen von Bereichen zu beachten, die Blockgrenzen überschreiten. Damit können Sie in einem letzten Schritt das RSF-Dateisystem in das VFS von Ulix integrieren, also einen RSF-Datenträger mounten und regulär über `u_open`, `u_read` etc. nutzen: Passen Sie die `u_`-Funktionen des VFS so an, dass neben den Minix-Funktionen `mx_*` nun auch die neuen RSF-Funktionen `rsf_*` aufgerufen werden.

## W 6 **. Prozess-Priorisierung (20 P.)

Unter Ulix sind alle Prozesse bzw. Threads gleichwertig, es gibt keine Priorisierung. In dieser Aufgabe implementieren Sie Prioritäten.

**a)** Ergänzen Sie im Thread Control Block einen „nice value“ (`int nice`), der Werte zwischen -20 und 19 annehmen kann. Der Standardwert ist 0. Prozesse vererben via `fork()` ihren Nice-Wert an Kindprozesse.

**b)** Schreiben Sie eine Funktion `int u_setpriority (int nice)`, die im laufenden Prozess den Nice-Wert auf den angegebenen Wert ändert. Ergänzen Sie passende Funktionen `syscall_setpriority` im Kernel sowie `int setpriority (int nice)` in der User-Mode-Bibliothek. (Der Rückgabewert von `u_setpriority` bzw. `setpriority` ist der neue Nice-Wert. Die Signatur entspricht nicht der von `setpriority()` unter Linux, ein Blick in die Manpage unter Linux ist trotzdem hilfreich.

**c)** Passen Sie im Code Chunk *<timer tasks>* den Code-Block

```
if (system_ticks % 2 == 0) {
  [...]
  scheduler (r, SCHED_SRC_TIMER);  // defined in the process chapter
  [...]
```

so an, dass nicht einfach bei jedem zweiten Timer-Interrupt der Prozess gewechselt wird, sondern dass hier der Nice-Wert des aktuellen Prozesses (`thread_table[current_task].nice`) berücksichtigt wird – im Ergebnis soll ein Prozess mit kleinerem Nice-Wert eine längere Zeitscheibe erhalten als einer mit größerem Nice-Wert.

**d)** Schreiben Sie ein Testprogramm, mit dem Sie überprüfen können, dass die Nice-Values wirklich vom System berücksichtigt werden.

## W 7 **. atexit(): Programmende konfigurieren (20 P.)

Unix-Systeme geben Prozessen die Möglichkeit, mit `atexit()` eine Funktion anzugeben, die beim Verlassen des Programms mit `exit()` oder `return` (aus `main()` heraus) aufgerufen wird. Implementieren Sie `atexit()` unter Ulix. Sie müssen dazu den Thread Control Block um ein neues Feld erweitern, das die Einsprungadresse einer Funktion speichern kann. Hinweise zur Funktion von `atexit()` können Sie der Linux-Manpage zu `atexit` entnehmen.

Im Rahmen dieser Aufgabe müssen Sie eine Kernel-Funktion `u_atexit()`, einen Syscall-Handler `syscall_atexit()` sowie eine User-Mode-Bibliotheksfunktion `atexit()` schreiben und die Funktion `syscall_exit()` anpassen.

Eine alternative Möglichkeit ist, die User-Mode-Bibliotheksfunktion `exit()` in `_exit()` umzubenennen und eine neue Funktion `exit()` in die Bibliothek aufzunehmen, die erst eine eventuell registrierte `atexit`-Funktion und dann `_exit()` aufruft.

Unter Linux & Co. ist es möglich, `atexit()` mehrfach aufzurufen, beim Programmende werden dann mehrere registrierte Funktionen aufgerufen. Das ist für diese Aufgabe nicht nötig: Eine `atexit`-Funktion reicht.

Schreiben Sie ein kleines Testprogramm, mit dem Sie Ihre Implementation testen.

---

*Viel Erfolg und vor allem: VIEL SPASS!*